Stateless: an Optic Algebra Library

Jesús López-González
j.lopezgo@alumnos.urjc.es

Juan Manuel Serrano
juanmanuel.serrano@urjc.es

Javier Fuentes
javier.fuentes@hablapps.com

June 22, 2017

Abstract

Abstract goes here!

1 Introduction

Software industry as we know it lacks well-founded frameworks to program reactive systems in a modular way. This kind of systems hide an inner *state* to their environment, showing therefore a black-box behaviour. Take a university information system as a guiding example. Universities are structured in departments, and in turn, each department has a budget. Even this simple definition does contain three kinds of entities: universities, departments and budgets. Ideally, programming the evolving state of the whole system should be achieved by programming the state of each entity or subsystem, and finally composing them all.

Provide solid references!

In addition, composition should contemplate heterogeneity: departments could be stored in a local database while budget could be deployed as external microservices. Ultimately, any configuration should be supported, but it is essential for the system description to remain pure and decoupled from low-level details, irrelevant to the domain. Needless to say that it should never hamper optimal performance. Programming a declarative and modular system in this way is still a great challenge, but functional programming is a good place to start [1].

There are plenty of technologies to develop reactive systems, but they are postulated as solutions to specific problems, and consequently, difficult to be reused. Turning to the university example, we could have decided to program department evolution on a relational database, like *MySQL*. However, if larger volumes of data arose in time, migrating to a distributed database such as *Redis* might become necessary. Given that situation, we would find it extremely difficult to adapt existing encodings to the new setting.

Fortunately, industry has detected this issue and new abstract tools have emerged to fill the gap. In this sense, we could have avoided the aforementioned problem by programming department evolution with *Slick*, a Functional-Relational Mapper that provides an *embedded domain specific language* [2] that can be later interpreted into different databases. Even though this language is a great help in this scenario, it is still quite limited to deal with database technologies. What if we change our mind and decide to implement department as a microservice?

We claim that many of these languages to access state could be further factorized, by providing a new language to unify them, while retaining reasonable levels of performance for a large class of applications. Moreover, we claim that the fundamentals behind *optics* —being *lens* [3] its

main representative — are ideal to abstract technology-specific details away. However, prior to that, they have to be taken to a more general setting: *optic algebras*.

Optic algebras distill the algebraic essence of an optic. To better understand this concept, we need to see optics from a coalgebraic perspective [4]. In this view, optics are just machines —that can possibly run forever [5]— with buttons that can be pressed to observe or forward the machine state [6]. In fact, what we mean by the algebraic essence is just the description of the machine interface in terms of its buttons, where any reference to the inner state has been abstracted away. The resulting interfaces are what we know as optic algebras and are postulated as the building blocks to program the state of our systems in a modular way.

Since composability is one of the most striking features of optics [7], one could imply similar abilities in their algebraic counterparts. However, as we will see in further sections, optic algebras in their *raw* representation do not compose as expected. In order to avoid that problem, we propose an alternative *natural* representation —since it relies on natural transformations [8]— that enables composition, in a similar way as for plain optics. Furthermore, this representation is key to introduce the notion of heterogeneity.

Optic algebra instances for the most widespread deployments should be provided. Given this scenario, by defining our system in terms of optic algebras, we should be able to get our logic implemented once and for all. Later, we could find it interpreted into a method call, a database access, a microservice invocation, or even as a combination of many of them.

This paragraph is awful, it should contain more insights on instances!

We claim that optic algebras in their natural representation lead to modular software designs. For instance, a traversal algebra could establish the connection between the university and each department. On the other hand, a lens algebra could describe the relationship between departments and budget. Then, we could compose both algebras to conform the whole system. Additionally, optic algebras will enable us to do it in an heterogeneous way, with several free instances ready to go, offering a reasonable performance for the majority of cases.

The rest of the paper is organized as follows.

Fulfill this paragraph when organization becomes clear

2 Background

2.1 Monads

2.2 Optics

2.2.1 Lens

```
trait Lens[S, A] {
  def get(s: S): A
  def set(a: A): S => S
}
```

$$set(get(s))(s) = s \qquad \textbf{GetPut}$$
$$get(set(a)(s)) = a \qquad \textbf{PutGet}$$
$$set(a2)(set(a1)(s)) = set(a2)(s) \qquad \textbf{PutPut}$$

2.2.2 Getter

```
trait Getter[S, A] {
  def get(s: S): A
```

```
}
```

2.2.3 Setter

```
trait Setter[S, A] {
  def modify(f: A => A): S
}
```

$$modify(identity)(s) = s \qquad \textbf{ModId}$$
$$modify(g)(modify(f)(s)) = modify(gcomposef)(s) \qquad \textbf{ModComp}$$

2.2.4 Fold

```
trait Fold[S, A] {
  def getList(s: S): List[A]
}
```

2.2.5 Traversal

For a fixed N

```
trait Traversal[S, A] {
  def getList(s: S): List[A]
  def setList(as: List[A]): S => S
}
```

$$setList(getList(s))(s) = s \qquad \textbf{GetPut}$$
$$getList(setList(as)(s)) = as \qquad \textbf{PutGet}$$
$$setList(as2)(setList(as1)(s)) = setList(as2)(s) \qquad \textbf{PutPut}$$

2.2.6 Affine Traversal

```
trait Affine[S, A] {
  def getOption(s: S): Option[A]
  def set(a: A): S => S
}
```

This laws makes no sense, take a look at haskell file

$$getOption(s).map(a => set(a)(s)) = s \qquad \textbf{GetSet}$$
$$getOption(set(a)(s)) = getOption(s).as(a) \qquad \textbf{SetGet}$$
$$set(a2)(set(a1)(s)) = set(a2)(s) \qquad \textbf{SetSet}$$

2.2.7 Prism

```
trait Prism[S, A] {
  def getOption(s: S): Option[A]
  def reverseGet(a: A): S
}
```

$$getOption(s).map(reverseGet) = s \qquad \textbf{GetRev}$$
$$getOption(reverseGet(a)) = Some(a) \qquad \textbf{RevGet}$$

2.2.8 Iso

```
trait Iso[S, A] {
  def get(s: S): A
  def reverseGet(a: A): S
}
```

$$reverseGet(get(s)) = s \qquad \textbf{GetRev}$$
$$get(reverseGet(a)) = a \qquad \textbf{RevGet}$$

2.3 Machines

3 Deriving Optic Algebras

Previously, in section 2.2, optics were shown as crucial abstractions to access certain state, particularly, they provide access to in-memory data structures. Furthermore, we discussed that this effect could be perfectly modeled with the *State Monad*. Thereby, optics enable programmers to describe a whole range of complex transformations as State programs. Having said so, it would be great to take this potential to other *stateful* scenarios.

Optic algebras distill optics to extract their algebraic essence. More specifically, they abstract all the in-memory specific details away, to make the optic potentially exploitable in new contexts, such as relational databases, microservices or event sourcing, among others. The remainder of this section will turn the most prominent optics into their corresponding algebraic representation, beginning with lens.

3.1 Lens Algebra

How could one extract the essence of a lens? What does it mean to abstract the in-memory details away? Well, we know that lenses can be represented as a pair of functions: *get* and *set*. Those functions are aware of the existence of an outer state, and they use it to extract the focus from, and to set a new value for the focus to. Our main objective here is to abstract away any reference to that outer state. We do so with the following steps:

Proof. Let s, a be types representing the whole and the focus of certain lens, respectively. So,

This proof is not a proof at all, lier!

(get, set)

$$
\begin{aligned}
&= (s \to a,\ a \to s \to s) && \text{(expand definitions)} \\
&= (() \to s \to a,\ a \to s \to s) && \text{(homogenize input)} \\
&= (() \to s \to a,\ a \to s \to (s,\ ())) && \text{(homogenize output)} \\
&= (() \to s \to (s,\ a),\ a \to s \to (s,\ ())) && \text{(homogenize resulting state *)} \\
&= (() \to State\ s\ a,\ a \to State\ s\ ()) && \text{(State isomorphism)} \\
&\therefore (() \to f\ a,\ a \to f\ ()) && \text{(Abstract away)}
\end{aligned}
$$

□

The first step after expanding definitions consists of making the input in both functions homogeneous. Since *set* receives an A as first parameter, something has to be added as a first argument in *get*. Thereby, by applying simple algebraic rules, we can safely prepend a redundant *Unit* typed parameter.

Secondly, we homogenize output. In this sense, *get* is returning an A as output, while *set* is only returning the updated value for S. If we want the latter to return its own output, we have to provide a redundant *Unit* accompanying the resulting state S.

Thirdly, once we have homogenized the output value, it should be desirable to homogenize the resulting state as well. To do so, we create a contrived S as companion of the output A, which will contain the second parameter as is. This step is tricky, since it is not a valid equivalence per se. Therefore, in order to formalize it, we have to impose that the resulting state has to be exactly the same as the incoming one, as a new law for our derived *get*.

Make this clear somehow in the corresponding step

Finally, it is not hard to see that the right hand side of the resulting functions corresponds with the state monad, being S the underlying state. Given this isomorphism, we can say we have our functions homogenized, both getting certain input and returning a stateful program. If you remember our initial objective, we wanted to abstract all those references to the outer state S away. Now, it is easy to do so, we just have to turn *State[S, ?]* into a generic type constructor *F[?]*. Hopefully, the resulting pair should be familiar to the reader.

We started our derivation trying to remove low-level details from our lens and we ended up with something really similar to *MonadState*. As you can see, we are lacking the monadic combinators to fully complete the connection. At first glance, this seems like a big gap, but you will be pleased to hear that we are not that far, we just need to bring lens laws into the picture.

Lens laws have an important role to connect the resulting algebra with *MonadState*. You must recall that a lens is not only a pair of functions, it is closely tied to the lens laws: *PutPut*, *GetPut* and *PutGet*. Therefore, they have to be taken into account in the derivation process, because they also mutate along with the pair of functions. Those adaptations are a walk in the park, but there is a particular point when the law adaptation no longer makes sense.

What does the *GetPut* law mean right after the last step? In fact, there is no way to establish a connection between the pair of functions at this level of f abstraction, unless we provide such mechanism with monadic combinators. Given this situation, it is reasonable to extend our algebra with *Monad*, because laws have to be adapted into this abstract setting. Luckily, we do not have to do such work, because this step has already be done. In fact, by adapting lens laws to the new monadic setting, we are just reinventing *MonadState* laws.

The lens and *MonadState* laws correspondence is straightforward, however *MonadState* has an additional intruder which is not contemplated in the lens laws: *GetGet*. So, we could think that *MonadState* is imposing additional restrictions to the ones that we adapted from lens laws. Nevertheless, we should recall that we added an additional law to our representation when we homogenized the resulting state. If we adapt it to the abstract setting, we end up with somethink

similar to this:

$$get >> return\ () = return\ () \qquad \text{(XGet)}$$

It tells us that instructing a *get* and ignoring its result does not produce an overall effect. It would be great if *GetGet* could be derived from *XGet*. We show that this is possible:

Proof. GetGet can be derived from XGet

Any leap of faith in this process?

Monad laws steps are quite abrupt

GetGet

$$\begin{aligned}
&= get >>= (a1 \to get >>= (a2 \to return\ (a1, a2))) = \\
&\quad get >>= (a \to return\ (a, a)) && \text{(GetGet)} \\
&= get >>= (a1 \to get >>= (a2 \to return\ (a1, a2))) >> return\ () = \\
&\quad get >>= (a \to return\ (a, a)) >> return\ () && \text{(Monad laws)} \\
&= get >>= (a1 \to get >>= (a2 \to return\ ())) = get >>= (a \to return\ ()) && \text{(Monad laws)} \\
&= get >> get >> return\ () = get >> return\ () && \text{(Syntax)} \\
&= get >> return\ () = get >> return\ () && \text{(XGet)} \quad \square
\end{aligned}$$

By establishing this connection we can say that the adaptation of lens laws, including the one that emerged in the abstraction process, corresponds directly with *MonadState* laws. Thereby, we claim that the algebraic essence of a lens is exactly *MonadState*. In fact, this typeclass abstract away all those in-memory specific details while preserving the lens laws.

3.2 Getter Algebra

Next, we will derive the optic algebra associated to *getter*. In fact, it is not hard to notice that this optic is a simplification of lens, where we are only interested in the *getting* capabilities. The derivation process to obtain a getter algebra is therefore:

Proof. Let s, a be types representing the whole and the focus of certain getter, respectively. So,

$$\begin{aligned}
&get \\
&\quad = s \to a && \text{(expand definitions)} \\
&\quad = () \to s \to a && \text{(homogenize input)} \\
&\quad = () \to s \to (s,\ a) && \text{(homogenize resulting state *)} \\
&\quad = () \to State\ s\ a && \text{(State isomorphism)} \\
&\quad \therefore () \to f\ a && \text{(Abstract away)} \quad \square
\end{aligned}$$

As one can see, the process is exactly the same as the one we applied over the first component of the pair of functions that represent a lens. What are the laws associated to this algebra? Since getter has no attached laws, the only one that should be associated to the new abstraction is *XGet*, which emerged while homogenizing the resulting state.

Is this algebra already known in the folklore? At first glance, we could think that *MonadReader* is the closest one to match. However, this typeclass declares an additional method *local* and consequently, new laws are required. It seems that we only need part of the definition of *MonadReader*. Indeed, some functional programming libraries do reflect this separation, resulting in the *MonadAsk* typeclass[1], which perfectly fulfills our algebra. As expected, it has a unique law, which corresponds with our *XGet*.

[1] https://github.com/purescript/purescript-transformers/blob/master/src/Control/Monad/Reader/Class.purs#L16

3.3 Setter Algebra

Setter ignores the lens getting part and focuses on the setting one. As we saw previously, it is slightly more powerful, since it is defined in terms of a *modify* method. As usual, we show now the derivation process:

Proof. Let s, a be types representing the whole and the focus of certain setter, respectively. So,

$$
\begin{array}{ll}
set & \\
\quad = (a \to a) \to s \to s & \text{(expand definitions)} \\
\quad = (a \to a) \to s \to (s,\ ()) & \text{(homogenize output)} \\
\quad = (a \to a) \to State\ s\ () & \text{(State isomorphism)} \\
\quad \therefore (a \to a) \to f\ () & \text{(Abstract away)}
\end{array}
$$

□

These steps do not generate new laws, so our resulting setter algebra laws will be just an adaptation of the setter optic ones. Since we have not found this algebra in the literature[2], we propose the next laws:

$$
\begin{array}{ll}
modify\ id = return\ () & \textbf{ModId} \\
modify\ f >> modify\ g = modify\ (g\ .\ f) & \textbf{ModComp}
\end{array}
$$

3.4 Fold Algebra

Fold algebras slightly modify getter algebras, returning a list of as instead of a single one. Thereby, derivation process is almost identical:

Proof. Let s, a be types representing the whole and the focus of certain fold, respectively. So,

$$
\begin{array}{ll}
getList & \\
\quad = s \to [a] & \text{(expand definitions)} \\
\quad = () \to s \to [a] & \text{(homogenize input)} \\
\quad = () \to s \to (s,\ [a]) & \text{(homogenize resulting state *)} \\
\quad = () \to State\ s\ [a] & \text{(State isomorphism)} \\
\quad \therefore () \to f\ [a] & \text{(Abstract away)}
\end{array}
$$

□

As far as we are concerned, no mainstream typeclass matches the expected algebra. However, taking getter algebra law as template, fold algebra one is straightforward.

[2]There are references to *MonadPut* though.

3.5 Affine Traversal Algebra

Proof. Let s, a be types representing the whole and the focus of certain affine traversal, respectively. So,

$(getMaybe, set)$

$$
\begin{aligned}
&= (s \to Maybe\ a,\ a \to s \to s) && \text{(expand definitions)}\\
&= (() \to s \to Maybe\ a,\ a \to s \to s) && \text{(homogenize input)}\\
&= (() \to s \to Maybe\ a,\ a \to s \to (s,\ ())) && \text{(homogenize output)}\\
&= (() \to s \to (s,\ Maybe\ a),\ a \to s \to (s,\ ())) && \text{(homogenize resulting state *)}\\
&= (() \to State\ s\ (Maybe\ a),\ a \to State\ s\ ()) && \text{(State isomorphism)}\\
&\therefore (() \to f\ (Maybe\ a),\ a \to f\ ()) && \text{(Abstract away)}
\end{aligned}
$$

□

Laws for emerging algebra:

$getMaybe >>= (maybe\ (return\ ())\ set) = return\ ()$	**GetmSet**
$set\ a >> getMaybe = getMaybe\ \$ > a$	**SetGetm**
$set\ a1 >> set\ a2 = set\ a2$	**SetSet**
$getOption >> return\ () = return\ ()$	**GetmGetm**

3.6 Traversal Algebra

Not a traversal, maybe we can call it *Weak Traversal*

Proof. Let s, a be types representing the whole and the focus of certain traversal, respectively. So,

$(getList, modify)$

$$
\begin{aligned}
&= (s \to [a]\,,\ (a \to\ a) \to s \to s) && \text{(expand definitions)}\\
&= (() \to s \to [a]\,,\ (a \to\ a) \to s \to s) && \text{(homogenize input)}\\
&= (() \to s \to [a]\,,\ (a \to\ a) \to s \to (s,\ ())) && \text{(homogenize output)}\\
&= (() \to s \to (s,\ [a]),\ (a \to\ a) \to s \to (s,\ ())) && \text{(homogenize resulting state *)}\\
&= (() \to State\ s\ [a]\,,\ (a \to\ a) \to State\ s\ ()) && \text{(State isomorphism)}\\
&\therefore (() \to f\ [a]\,,\ (a \to\ a) \to f\ ()) && \text{(Abstract away)}
\end{aligned}
$$

□

Laws for emerging algebra:

GetlMod makes no sense, it would have been different if we had *setList* available

These laws are our own! Remember, this isn't a standard Traversal.

$modify\ id = return\ ()$	**ModId**
$modify\ f >> modify\ g = modify\ (g\ .\ f)$	**ModMod**
$modify\ f >> getList = getList >>!\ modify\ f >>= (return\ .\ map\ f)$	**ModGetl**
$getList >> return\ () = return\ ()$	**GetlGetl**

3.7 Other Optic Algebras

Iso & Lens share algebra. Prism & Affine share algebra.

What does this bring us?

Is Iso really an asymmetric optic? (connections with symmetric are obvious)

4 Natural Composition and Heterogeneity

Once our optic algebras are defined, we could be interested in combining them, just as plain optics are combined. However, when we move to this scenario, the situation becomes more complicated. As we know, lenses compose when the *focus* of the first lens matches the *whole* of the second one. However, our optic algebras have a type constructor as weird *whole* and a concrete type as *focus*, which leads to an odd scenario.

Therefore, we need to ask ourselves what does it mean to compose a pair of optic algebras. For instance, suppose we want to compose *TraversalAlg f Department* with *LensAlg g Int*, as we showed in the university example. The first one knows how to build f programs to access all the departments. On the other hand, the second instance builds g programs to access the budget. If we think of f and g as something encapsulating the university and the department states, respectively, we could determine that the composition of both algebras should be close to *TraversalAlg f Int*, ie. a program over the whole university that has access to all the budgets.

Unluckily, the aforementioned composition is not possible, since additional components are required to implement it. Particularly, a natural transformation $q \rightsquigarrow p$ would serve us to embed focus programs into whole ones. Thereby, optic algebras cannot be composed as we compose plain optics, in an autonomous way.

5 Instances for Free

6 Conclusions and Future Work

7 Acknowledgements

References

[1] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[2] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[3] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[4] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59. Cambridge University Press, 2016.

[5] David A Turner. Total functional programming. *J. UCS*, 10(7):751–768, 2004.

[6] Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.

[7] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *arXiv preprint arXiv:1703.10857*, 2017.

[8] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.